INFO-H-515 Project 2022–2023, Part I

2022 - 2023

Massa ALSAFADI MAHMALJI

Oussama MIFDAL

Ali Manzer

Sami Abdul Sater

Group-6

Introduction

The advent of big data has introduced datasets of such vast size that conventional database software tools struggle with storage, management, and analysis. The goal of this project is to use distributed data management techniques to perform correlation analysis over the Brussels Mobility Bike Counts dataset. This report details our implementation of all the tasks, providing explanations for our specific choices and methods.

Data Exploration

Data from the Brussels Mobility Bike Counts API is processed using PySpark framework in Python. The devices are sensors located in Brussels that count the number of bikes that pass by and their average speed. Using the public API, we downloaded the historical data from, for the 18 different sensors.

First, we download the list of all the unique sensors from the API, then for each one of them we request their data from 2018-12-06 to 2023-03-31, they are all stored in a single dataframe, which has the following features: time stamp, count, speed and device name.

```
                 day  tgap  Count  Speed DeviceName
0         2018-12-06    51      2     20     CB2105
1         2018-12-06    52      3     27     CB2105
2         2018-12-06    53      3     17     CB2105
3         2018-12-06    54      2     24     CB2105
4         2018-12-06    55      1     18     CB2105
5         2018-12-10    56      6     18     CB2105
6         2018-12-10    57      1     14     CB2105
7         2018-12-10    58      7     17     CB2105
8         2018-12-10    59      5     21     CB2105
9         2018-12-10    60      7     20     CB2105
10        2018-12-10    95      1     18     CB2105
```

Our first observation is that, as described in the assignment, some sensors have no data for when no bikes have passed in front of them, these data points are added back with the missing values of Count = 0, and average speed = -1. Secondly, we noticed that not all sensors have data spanning the entire selected period, indicating that some sensors were added later on, we decided to add data even for the periods where they weren't installed by simply adding the same missing values to all of these data points. One other way this could have been tackled is by using the mean values of Count and Average Speed instead.

As part of the data processing, we combine the Day and tgap columns into one single tgap column, where the values for the second day continue from 97 to 192 and so on (essentially, tgap becomes an index). To fill in the missing values we use the **reindex** method in pandas to create the missing tgap rows with the same missing values as above for Count and Average speed.

Task 1: Batch Processing

The Pearson Correlation Coefficient (PCC)is defined as:

$$r_{ij}(t) = \frac{\sum_{n=1}^{t}(c_i(n) - \bar{c_i}(t))(c_j(n) - \bar{c_j}(t))}{\sqrt{\sum_{n=1}^{t}(c_i(n) - \bar{c_i}(t))^2}\sqrt{\sum_{n=1}^{t}(c_j(n) - \bar{c_j}(t))^2}},$$

The task is about computing analytics over the entire dataset, which requires filtering, aggregating, joining, and sorting operations, so we chose the **Spark Data Frame** structure, more suitable for aggregated queries than the RDD. To compute the PCC, we rely on the pyspark.sql.Window object and pyspark.sql.functions for the computations(the sum of a column with F.sum). For each row, a window will be defined, partitioned over the "DeviceName"(the analytics are computed for each device separately), ordered by tgap, and in its rangeBetween function, the first parameter, the lower bound, will be the first row of the data frame, meaning the window takes all rows before the current one, the upper bound parameter will be zero, so that no other rows after the time gap is considered.

First, we compute the moving average, by computing for each row the cumulative count, we then divide it by the time gap t.

$$\bar{c_i}(t) = 1/t \sum_{n=0}^{t} c_i(n)$$

Followed by the standard deviation, which required using a second window to keep the average c_i value constant when doing the c_i(n) – c̅_i (t) calculation,

$$\sigma_i(t) = \sqrt{\sum_{n=0}^{t}[c_i(n) - \bar{c_i}(t)]^2}$$

We then create all the possible pairs with a join on time gaps, without duplicates(s1_Name < s2_Name. For the covariance, for each row (each pair, each time gap), we now compute it, with a similar technique as for the standard deviation as defined below

$$\sigma_{i,j}(t) = \sum_{n=0}^{t}(c_i(n) - \bar{c_i}(t)) \cdot (c_j(n) - \bar{c_j}(t))$$

Finally for PCC between two sensors i and j is defined for each time gap as

$$r_{ij}(t) = \frac{\sigma_{i,j}(t)}{\sigma_i(t) \cdot \sigma_j(t)} \ ,$$

where the σ_i is the moving standard deviation for each sensor as computed above, and the covariance $\sigma i, j$ is computed as above as well. Finally for the top 5 correlated pairs we partition by tgap and order by PCC and for each tgap, we associate the rank to each row and only keep top 5 rows as shown below.

tgap	s1_Name	s2_Name	pearsonCoeff
151391	CB2105	CJM90	0.8352454561723874
151391	CB2105	CEK049	0.8172815203063362
151391	CEK049	CJM90	0.8133373740042474
151391	CB1599	CEK049	0.8088956581175814
151391	CAT17	CVT387	0.7985076772625357
151390	CB2105	CJM90	0.8352459639225993
151390	CB2105	CEK049	0.8172827939849917
151390	CEK049	CJM90	0.8133371912720383
151390	CB1599	CEK049	0.8088952868085363
151390	CAT17	CVT387	0.7985073661930835
151389	CB2105	CJM90	0.8352457135708057
151389	CB2105	CEK049	0.8172825203389336
151389	CEK049	CJM90	0.8133368503200527
151389	CB1599	CEK049	0.8088949827441813
151389	CAT17	CVT387	0.7985072370714433
151388	CB2105	CJM90	0.835245472079438
151388	CB2105	CEK049	0.8172822510309377
151388	CEK049	CJM90	0.8133365203973153
151388	CB1599	CEK049	0.8088944294256768
151388	CAT17	CVT387	0.7985071315380092

only showing top 20 rows

Task 2: Stream Processing

Data is not only big, but it is also fast by arriving from devices and API’s every seconds so it needs real time analysis instead of batch analysis. Therefore, we will use streaming data framework from Spark to analyze this project data. In this section streaming processing will be implemented on mini batches with fixed duration and amount of data sent from the producer to consumer. Here we will create a producer notebook to read the data and at each time interval, sending some data to a server. This requires a connection to the server first to connect the consumer notebook to create a socket of data messages. The parameters are the size of the batch `pi`, i.e the number of days of data that will be sent each time, and `delta`, the time interval. In this scenario, we will consider a huge spark dataframe from which we will send the data. Each time the `delta` period is elapsed, a batch of data is emitted, and this batch is a subset of the dataframe and will be saved as an RDD. This subset is built around the `timegap` value of the rows, for which at each emission we set a lower bound and an upper bound, and all the data between those bounds is selected. For instance, the bounds for the first iteration are obviously `0` and `gaps_per_day*pi`. For the next iteration, the lower bound becomes the former upper bound `+1`, and the upper bound becomes the current lower bound `+gaps_per_day*pi`. The minibatches are encoded in an RDD so we can use the filter method to send messages between upper and lower bound. Now after sending the mini batches, the consumer will calculate the PCC as done in batch processing. In our example we chose pi to be 5 days and delta as 10 seconds so each minibatch will be send as an RDD for 5 days * 96 time gaps= 480 data lines and calculate the top 5 correlated pairs of sensors. The figures below show the second and the third mini batch starts at 480+48=960

tgap	s1_Name	s2_Name	pearsonCoeff
479	CB2105	COM205	0.922960860059877
479	CEK049	CJM90	0.9195009569801788
479	CEK049	COM205	0.912215215741282
479	CJM90	COM205	0.8925394861813455
479	CB2105	CJM90	0.7980190090628887
478	CB02411	COM205	0.958077244282212
478	CB02411	CEK049	0.942149925999186
478	CB02411	CJM90	0.9373907221525857
478	CB2105	COM205	0.9255241663746389
478	CEK049	CJM90	0.9199400732754083
477	CB02411	COM205	0.9580756586596386
477	CB02411	CEK049	0.9421477654057824
477	CB02411	CJM90	0.9391747269513185
477	CB2105	COM205	0.9255216360302251
477	CEK049	CJM90	0.9216870248580247
476	CB02411	COM205	0.9580740662610626
476	CB02411	CJM90	0.940192276775478
476	CB2105	COM205	0.9255190948920032
476	CEK049	CJM90	0.9226825715037834

only showing top 20 rows

tgap	s1_Name	s2_Name	pearsonCoeff
960	CB2105	CEK049	0.7352982958517671
960	CB2105	CJM90	0.7062163116722353
960	CEK049	CJM90	0.6979926644167486
960	CEK049	COM205	0.6433196269172173
960	CB1143	CEK049	0.6336139537705212

Task 3: Sliding Window Processing

For this third task, we are using Spark Streaming’s built-in sliding window functionality. The objective here is to see the evolution of the pearson correlation coefficient over the data received in the last W seconds, this W represents the length of the window, and it is moved every Δ seconds. To make sure that the window doesn’t not miss any data the W needs to be a multiple of Δ. For example if we send a batch of data every 5 seconds from the producer, on the consumer side, with a W of 30 seconds, the window will contain 6 batches worth of data(except at the very beginning and when the producer stops sending, in which case the window empties out progressively). For this task we take in the incoming data in the form of RDD, and inside the window function we create a spark Dataframe structure, which spark inherently optimizes in a distributed manner, and then apply the compute_analytics() function which is essentially what is done in the batch processing(here we consider the content of the window as the batch).

```
+----+-------+-------+------------------+
|tgap|s1_Name|s2_Name|      pearsonCoeff|
+----+-------+-------+------------------+
|  99|CB02411| COM205|0.9767055043570113|
|  99|CB02411| CEK049|0.9675249810012775|
|  99|CB02411|  CJM90|0.9647024820705544|
|  99| CB2105| COM205|0.9562230318359805|
|  99| CEK049|  CJM90|0.9544551444700105|
|  98|CB02411| COM205| 0.976694090139714|
|  98|CB02411| CEK049|0.9675091130410687|
|  98|CB02411|  CJM90|0.9646850889643144|
|  98| CB2105| COM205|0.9562028399563188|
|  98| CEK049|  CJM90|0.9544329959738785|
|  97|CB02411| COM205|0.9766824302519685|
|  97|CB02411| CEK049|0.9674929035962033|
|  97|CB02411|  CJM90|0.9646673214042067|
|  97| CB2105| COM205|0.9561822146733346|
|  97| CEK049|  CJM90|0.9544103709409724|
|  96|CB02411| COM205|0.9766705166761354|
|  96|CB02411| CEK049|0.9674763415235428|
|  96|CB02411|  CJM90|0.9646491671663954|
|  96| CB2105| COM205|0.9561611418817408|
|  96| CEK049|  CJM90|0.9543872538247541|
+----+-------+-------+------------------+
only showing top 20 rows

+----+-------+-------+------------------+
|tgap|s1_Name|s2_Name|      pearsonCoeff|
+----+-------+-------+------------------+
|  99|CB02411| COM205|0.9767055043570113|
|  99|CB02411| CEK049|0.9675249810012775|
|  99|CB02411|  CJM90|0.9647024820705544|
```

```
+----+-------+-------+------------------+
|tgap|s1_Name|s2_Name|      pearsonCoeff|
+----+-------+-------+------------------+
|  99|CB02411| COM205|0.9767055043570113|
|  99|CB02411| CEK049|0.9675249810012775|
|  99|CB02411|  CJM90|0.9647024820705544|
|  99| CB2105| COM205|0.9562230318359805|
|  99| CEK049|  CJM90|0.9544551444700105|
|  98|CB02411| COM205| 0.976694090139714|
|  98|CB02411| CEK049|0.9675091130410687|
|  98|CB02411|  CJM90|0.9646850889643144|
|  98| CB2105| COM205|0.9562028399563188|
|  98| CEK049|  CJM90|0.9544329959738785|
|  97|CB02411| COM205|0.9766824302519685|
|  97|CB02411| CEK049|0.9674929035962033|
|  97|CB02411|  CJM90|0.9646673214042067|
|  97| CB2105| COM205|0.9561822146733346|
|  97| CEK049|  CJM90|0.9544103709409724|
|  96|CB02411| COM205|0.9766705166761354|
|  96|CB02411| CEK049|0.9674763415235428|
|  96|CB02411|  CJM90|0.9646491671663954|
|  96| CB2105| COM205|0.9561611418817408|
|  96| CEK049|  CJM90|0.9543872538247541|
+----+-------+-------+------------------+
only showing top 20 rows
```

Conclusion

To conclude, we can process big data easily and more efficiently by partitioning datasets and distributing them to nodes in Spark clusters. This partitioning strategy enables workload distribution and parallel execution across the cluster, enhancing overall system performance. It also provides enormous variety of operations that can be done on dataframes then can be collected again to a single one. Spark also provides streaming partitions of the input stream into disjoint time intervals for real time computations. The data in each time interval becomes a (mini-batch) RDD. Therefore, normal spark operators can be used to transform/act on mini-batch RDDs. In terms of scalability the system is designed to be highly scalable for growing data accommodate growing workloads. Thus the system can scale by adding more sensors and nodes to clusters so parallelism of the processing across a larger number of nodes can occur.

Link to our video **: https://www.youtube.com/watch?v=C1BIHeqTWnw or https://www.dropbox.com/s/9tydzzxzmjb24mp/Phase_1.mov?dl=0**

References

[1] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#creating-streaming-dataframes-and-streaming-datasets

[2] https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/

[3] https://stackoverflow.com/questions/59240277/insert-missing-date-rows-and-insert-old-values-in-the-new-rows-pyspark

[4] Spark lecture notes